나무위키

TLS 핑거프린팅

최근 수정 시각: 2026-08-02 11:54:37

분류: 인터넷 | 프라이버시 | 나무위키 프로그래밍 프로젝트

목차

1. 개요
2. 원리
   2.1. 특정성
3. 표준
4. 쓰임
5. 식별 우회 방법
6. 관련 문서

1. 개요

TLS를 통한 통신시 TLS 클라이언트나 서버의 패킷 패턴을 파악하여 상대를 특정하는 기술.

본 문서는 클라이언트 쪽의 fingerprinting을 주로 다룬다.

2. 원리

RFC 5246 및 RFC 8446(1.3) 및 SF JA3 기준으로 설명한다.

기본적으로 TLS client에서 key generation을 마치고 server에 실제 접속 시 보내는 최초 패킷을 `ClientHello`라고 부르는데, RFC 문서를 보면

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

요렇게 생겼다. 왜 굳이 ClientHello인가 싶을 수 있는데, 해당 패킷은 key exchange가 되기 전, 즉 암호화되지 않은 평문으로 보내지는 값이기 때문.

JA3 기준 여기서 쓰이는 데이터는

1. `SSLVersion` (`client_version`)
2. `Cipher` (`cipher_suites`)
3. `SSLExtension` (`extensions`) (있다면)
4. `EllipticCurve` (있다면)
5. `EllipticCurvePointFormat` (있다면) - RFC 4492

이렇게 다섯 종류인데, 없는 건 그냥 비워도 된다. 각 field는 정확히 저 순서대로 나열해서 ,로 구분하고, cipher_suites처럼 배열로 오는 것들은 -로 구분한다.

실제 2026년 최신 버전 구글 Chrome으로 한번 계산을 해보자. 후술하지만 아래 예시로 나온 fingerprint가 당신이 지금 쓰고 있는 Chrome의 fingerprint랑 겹칠 확률은 극히 낮으니 예시로만 보자.

1. `SSLVersion`: TLS 1.2 버전 `0x0303` (771_{10}) - 참고로 TLS 1.3을 써도 `ClientHello` 패킷은 무조건 1.2로 시작한다. ~~그야...~~ 때문에 JA3 기준 raw fingerprint는 사실상 거의 다 771로 시작한다. 사실상 엔트로피에 큰 도움이 안 되기 때문에 JA4 등 최신 표준의 경우 `supported_versions` 등의 정보를 사용하기도 한다.
2. `Cipher`
   1. `TLS_GREASE 0xcaca` (51914_{10})
   2. `TLS_AES_128_GCM_SHA256 0x1301` (4865_{10})
   3. `TLS_AES_256_GCM_SHA384 0x1302` (4866_{10})
   4. `TLS_CHACHA20_POLY1305_SHA256 0x1303` (4867_{10})
   5. `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 0xc02b` (49195_{10})
   6. `TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 0xc02f` (49199_{10})
   7. `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 0xc02c` (49196_{10})
   8. `TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 0xc030` (49200_{10})
   9. `TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 0xcca9` (52393_{10})
   10. `TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 0xcca8` (52392_{10})
   11. `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA 0xc013` (49171_{10})
   12. `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA 0xc014` (49172_{10})
   13. `TLS_RSA_WITH_AES_128_GCM_SHA256 0x009c` (156_{10})
   14. `TLS_RSA_WITH_AES_256_GCM_SHA384 0x009d` (157_{10})
   15. `TLS_RSA_WITH_AES_128_CBC_SHA 0x002f` (47_{10})
   16. `TLS_RSA_WITH_AES_256_CBC_SHA 0x0035` (53_{10})
3. `SSLExtension`
   1. `GREASE 0x3a3a` (14906_{10})
   2. `supported_groups 0x000a` (10_{10})
   3. `application_layer_protocol_negotiation 0x0010` (16_{10})
   4. `renegotiation_info 0xff01` (65281_{10})
   5. `compress_certificate 0x001b` (27_{10})
   6. `supported_versions 0x002b` (43_{10})
   7. `signed_certificate_timestamp 0x0012` (18_{10})
   8. `encrypted_client_hello 0xfe0d` (65037_{10})

9. `extended_master_secret 0x0017` (23_{10})
10. `psk_key_exchange_modes 0x002d` (45_{10})
11. `application_settings 0x44cd` (17613_{10})
12. `ec_point_formats 0x000b` (11_{10})
13. `key_share 0x0033` (51_{10})
14. `server_name 0x0000` (0_{10})
15. `session_ticket 0x0023` (35_{10})
16. `status_request 0x0005` (5_{10})
17. `signature_algorithms 0x000d` (13_{10})
18. `GREASE 0x9a9a` (39578_{10})

4. `EllipticCurve`
    1. `GREASE 0x8a8a` (35466_{10})
    2. `X25519MLKEM768 0x11ec` (4588_{10})~~이게 왜 EllipticCurve인지는 묻지 말자~~
    3. `x25519 0x001d` (29_{10})
    4. `secp256r1 0x0017` (23_{10})
    5. `secp384r1 0x0018` (24_{10})
5. `EllipticCurvePointFormat`
    1. `uncompressed 0x0` (0_{10})

직렬화에 앞서 여기서 GREASE는 모조리 거르는데, GREASE 특성상 시간에 따라 값이 유동적이니 GREASE를 쓰는 TLS 클라이언트에서도 같은 fingerprint가 나오도록 하기 때문이다.#

이걸 앞서 설명한 방식대로 직렬화하면
`771,4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53,10-16-65281-27-43-18-65037-23-45-17613-11-51-0-35-5-13,4588-29-23-24,0`
을 얻고, 이 raw fingerprint를 MD5로 해싱하면 다음과 같은 최종 fingerprint를 얻는다.
`0bee4257aeb31d51fe3bb401270e5074`

2.1. 특정성

식별성이 그렇게 아주 높다고 하기는 어렵다. 심지어 TLS 단에서 결정되는 내용이기에 쿠키나 유저 에이전트 등과 달리 애플리케이션 레이어 이상 고수준의 내용까지 파악하는 것은 힘들다. 가장 경우가 좋다면 사용중인 브라우저/클라이언트의 종류나 버전, OpenSSL 등 라이브러리의 버전 등을 추측할 수 있다는 것인데, 유저 에이전트와 마찬가지로 그 자체만으로는 개인정보가 아니나 다른 fingerprint와 조합되었을 때 개인 특정의 가능성이 훨씬 커질 수는 있다.

위 내용을 보면 list의 순서에도 영향을 받기 때문에 순서를 섞어서 위장하기도 하는데, 이를 해결하기 위해 해싱 전 lexical normalization을 하는 방법이 있다. 특히 JA4는 경우 normalization 과정이 아예 표준으로 포함되었다. 가령 위 예시를 normalize하면 `771,4865-4866-4867-49195-49199-49196-49200-52393-52392-49171-49172-156-157-47-53,0-5-10-11-13-16-18-23-27-35-43-45-51-17613-65037-65281,4588-29-23-24,0`에 `8e19337e7524d2573be54efb2b0784c9`가 된다.

> Using a fixed extension order can encourage server implementers to fingerprint Chrome and then assume specific implementation behavior.## #

가령 Chrome의 경우 2023년 110 버전부터 TLS ClientHello extension randomization 기능을 지원하기 시작했다.

이건 JA3 한정이지만 2026년 현재는 대부분의 브라우저에 JA3 회피 기술이 내장되어 있어 큰 위협이 되지 않는 수준이긴 하다. 물론 JA4 등 다른 fingerprint는 stable한 것도 많으므로 주의.

3. 표준

상술한 JA3 말고도 TLS 수준에서 동작하는 다양한 fingerprinting 표준이 존재한다.

- 클라이언트
    - JA3 - 현재는 세일즈포스에서 유지보수를 중단한 상태이며, 대부분 JA4로 넘어가는 추세이다. 여담으로 이름이 JA3인 이유는 해당 표준의 최초 개발자인 John Althouse, Jeff Atkinson, Josh Atkins# 세 명 모두 이니셜이 JA 꼴이기 때문이라고.
    - JA3N - JA3에 상술한 lexical normalization을 적용한 표준.
    - JA4
    - PeetPrint
- 서버
    - JA3S
    - JA4S

4. 쓰임

세일즈포스에서 처음 JA3/JA3S를 개발했을 당시 최초 목적은 멀웨어 탐지 및 fingerprinting이었다.

봇 차단 용으로도 쓰이는데, TLS fingerprinting이 적극 사용되는 대표적인 경우로 Cloudflare WAF 등이 있다. 단, WAF 단 커스텀 fingerprint block 설정시 엔터프라이즈 플랜이 필요하다.#

꼭 트래픽을 차단하지 않더라도 fingerprint나 식별된 브라우저에 따라서 다른 방향으로 라우트해주는 등 리버스 프록싱 용도로도 쓰일 수 있다.

5. 식별 우회 방법

~~사실 이 문서의 존재이유~~

상술했듯이 실제 브라우저의 fingerprint는 버전마다 정말 빠르게 바뀌므로 따라가기 벅찰 수 있고, 당연히 vendor들도 fingerprinting을 모르는 게 아니기에 다양한 기술을 적용해서 점점 난이도가 올라가는 편. 근본적으로 `ClientHello`를 통한 미세한 정보 공개는 TLS 프로토콜상 사실상 불가피한 영역이고, 최신 vendor 기술이 탑재된 브라우저를 쓰는 상황에서 노력한다고 딱히 크게 개선할 수 있는 것은 없으니 단순히 웹 브라우저 등을 사용하는 일반인이 이 문서를 보고 있다면 프라이버시에 대해선 별로 걱정할 필요가 없다.

반면 HTTP/클라이언트 등 특정 클라이언트는 소스 코드가 공개되어 있고 그 동작이 거의 정해져 있으며, 특성상 라이브러리들은 상용 브라우저들이 잘 지원하지 않는 온갖 확장과 구버전 기능을 폭넓게 지원하다 보니 상대적으로 특정이 쉬운데, 이런 경우 문제가 된다.

한때 permutation을 통한 우회가 통하던 시절이 있었으나 상술한 정규화로 대부분 막히기 때문에 확률은 높지 않은 편.

가장 효과적인 해결책은 다양한 TLS 클라이언트를 실행해 보고 그중 block되지 않은 것을 찾는 것이다(...). 무식해 보이지만 상술한 이유로 모든 버전의 모든 클라이언트를 전부 커버하는 fingerprint table을 구축하고 유지보수하기는 꽤나 힘들고, 후술할 방법을 찾는데 비용을 들이지 않고 빠르게 해결할 수 있는 방법이다.

물론 기술적으로 브라우저 fingerprint를 흉내내는 것은 어렵지 않으니 구현을 직접(~~음?~~) 하거나 구현체를 찾으면 된다. 가령 libcurl을 쓰고 있다면 `curl-impersonate`를 사용하는 등의 방법이 있다.

라이브러리 영향을 많이 받는 편인데, 가령 대표적으로 널리 쓰이는 OpenSSL의 경우 2026년 현재까지도 extension shuffling을 지원하지 않는다. #19220 반대로 대부분의 fingerprint 위조 구현체들은 OpenSSL을 포크하거나 대신 다른 라이브러리에 기반하는 편. 가령 OpenSSL 안쓰고 자체구현 `crypto/tls`를 쓰는 Go 등은 아예 언어 특정(...)이 될 수 있다는 단점이 있으나 자유도가 훨씬 높기 때문에 이를 활용한 uTLS와 httpcloak이 존재한다. uTLS 덕분에 우회 구현체들은 Go로 짜인 게 굉장히 많은 편.

6. 관련 문서

- 스푸핑

이 저작물은 CC BY-NC-SA 2.0 KR에 따라 이용할 수 있습니다. (단, 라이선스가 명시된 일부 문서 및 삽화 제외)
기여하신 문서의 저작권은 각 기여자에게 있으며, 각 기여자는 기여하신 부분의 저작권을 갖습니다.

나무위키는 백과사전이 아니며 검증되지 않았거나, 편향적이거나, 잘못된 서술이 있을 수 있습니다.
나무위키는 위키위키입니다. 여러분이 직접 문서를 고칠 수 있으며, 다른 사람의 의견을 원할 경우 직접 토론을 발제할 수 있습니다.